Extending the theory of Owicki and Gries with a logic of progress

Brijesh Dongol & Doug Goldson

School of ITEE,
University of Queensland,
Brisbane, Australia

January 12, 2005

Abstract

This paper describes a logic of progress for concurrent programs. The logic is based on that of UNITY, molded to fit a sequential programming model. Integration of the two is achieved by using auxiliary variables in a systematic way that incorporates program counters into the program text. The rules for progress in UNITY are then modified to suit this new system. This modification is however subtle enough to allow the theory of Owicki and Gries to be used without change.

1 Introduction

While verifying concurrent programs has been the topic of much research, deriving them has not. Even less work has been put into deriving concurrent programs in a way that gives equal consideration to both progress and safety requirements (as opposed to derivation that is based only on safety requirements). This paper contributes to this goal by defining a new logic of safety and progress. The paper does not address methodological questions of how to incorporate the logic into a design method for concurrent program derivation, and this is left as a subject for further work. The paper confines itself to defining the new logic, presenting an example of its use, and describing how the logic compares to other work in this area.

The point of departure for this paper is the theory of Owicki and Gries [OG76, Dij82, FvG99], a theory of partial correctness only, which means that it can only be used to reason about safety requirements. Two reasons recommend this point of departure. The first is that this theory is attractively simple. Proofs are carried out in a programming language (using the assertional style of Hoare) rather than in some other programming model such as a Petri net, IO automaton, or process algebra. We see this as an important advantage for program design, where the practicality of model-based reasoning turns, in some large part, on the transparency, ease and reliability of the translation of the model into code.

The second reason for using the theory is that it has already been used as an effective method of concurrent program derivation [FvG99], albeit derivation that is based only on safety requirements. The attitude of Feijen and van Gasteren is instructive in this regard, as it represents a deliberate decision to eschew the expressiveness of temporal logic in favour of the simplicity of Owicki and Gries. The benefit of doing so is a collection of design heuristics that are attractively simple to use and that, as already remarked, have been shown to be effective. The cost of the decision is that reasoning about progress requirements becomes both informal and post hoc. It is a welcome outcome that so much can be achieved in this way, yet it remains true that satisfaction of progress requirements using this approach is in an important sense left to chance. The pragmatic attitude of Feijen and van Gasteren, together with the limitation of the theory of Owicki and Gries, sets the methodological agenda for this paper. That is, the paper describes how to extend the theory of Owicki and Gries with a logic of progress that so far as possible, retains the simplicity of the original theory while at the same time provides a logic in which to formalise and

prove progress requirements. This work then is a prolegomenon to our larger goal, which is a method of program derivation that assigns equal consideration to both progress and safety requirements.

The step from standard predicate logic to temporal predicate logic represents an order of magnitude increase in complexity, which is why Feijen and van Gasteren refused to take it. In their words, "powerful formalisms for dealing with progress are available. However, the thing that has discouraged *us* from using them in practice is that they bring about so much formal complexity. ... We have decided to investigate how far we can get in designing multiprograms without doing *formal* justice to progress" ([FvG99] p79). Other authors, while taking the step, fully recognise its significance. For instance, Lamport writes "TLA differs from other temporal logics because it is based on the principle that temporal logic is a necessary evil that should be avoided as much as possible. Temporal formulas tend to be harder to understand than formulas of ordinary first-order logic, and temporal logic reasoning is more complicated than ordinary mathematical reasoning" ([Lam94], p917). Caution in the face of this added complexity has recommended to us the approach taken in UNITY [CM88], where assertion 'P leads to Q' formalises an important class of progress requirements called 'eventuality' requirements, and where eventuality assertions are defined without using temporal logic. The progress logic of UNITY is ideal for three reasons: the rules fully capture the temporal notion of leads-to [GP89], they thereby support reasoning about progress without resort to operational reasoning, and the rules are simple to use (relative to comparable program logics such as [Sch97, Lam94]). At the same time, we resile from the UNITY programming model because it lacks all notion of a control state, which makes (what should be simple) conventional sequential programming much harder. Fundamental operators such as ";" cannot easily be represented [SdR94].

So we have chosen to add the complexity of the logic of UNITY to the theory of Owicki and Gries over the complexity of full temporal predicate logic, or, to be more precise, to add a logic of progress that, while clearly inspired by UNITY logic, is tailored to fit the fundamentally different programming model of multiple sequential programs. In adapting the UNITY logic to fit a sequential programming model, a decision on how to represent the control state of a sequential program was first to be made. [OG76] offers a partial representation of control through the use of auxiliary variables, while [Sch97, Lam87] opt for a fuller representation through the use of control predicates. Our approach is a novel use of auxiliary variables to represent program counters, which provides a complete representation of the control points in a sequential program. This means that the formal complications that are introduced by the use of control predicates in the generalised Hoare logic of [Sch97, Lam87] are avoided in our logic, and we are able to retain the predicate transformer semantics of Dijkstra. The main contribution of this paper is to combine the strengths of these two different theories, Owicki-Gries and UNITY, in order to create something new.

The paper is structured as follows. Section 2 describes the theory of Owicki and Gries and provides background to Section 3 which gives the formal basis for the extended logic described in Section 4. An application of the new logic to a program design task is also given in Section 4 and finally Section 5 makes a conclusion.

2 The theory of Owicki and Gries

This section describes the theory of Owicki and Gries [OG76, Dij82, FvG99]. Section 2.1 describes the underlying programming language and its operational model. Section 2.2 describes the predicate transformer *wlp* that underlies the logical model of programs and concludes with the core theory of Owicki and Gries.

2.1 The programming language and its operational model

The programming notation is the language of guarded commands [Dij76].

Definition (*Statement*) For statements $S, S_1, S_2, \ldots S_n$, booleans $B_1, B_2, \ldots, B_n$, variables $x_1 \ldots x_m$ and expressions $E_1 \ldots E_m$, a *statement* is defined inductively as follows.

1. *skip* is a statement.
2. A (multiple) assignment $\overline{x{:=}\ E}$ is a statement where,

   $$\overline{x{:=}\ E} \mathrel{\widehat{=}} x_1{:=}\ E_1 \parallel x_2{:=}\ E_2 \parallel \ldots \parallel x_m{:=}\ E_m$$

   and $x_i \neq x_j$ for $i \neq j$.
3. $S_1;\ S_2$ is a statement.
4. $\langle S \rangle$ is a statement.
5. The following are statements, where each $B_i \rightarrow S_i$ is called a *guarded command* with *guard* B_i and *command* S_i.
   (a) **if** $B_1 \rightarrow S_1 \,[]\, B_2 \rightarrow S_2 \,[]\, \ldots \,[]\, B_n \rightarrow S_n$ **fi**
   (b) **do** $B_1 \rightarrow S_1 \,[]\, B_2 \rightarrow S_2 \,[]\, \ldots \,[]\, B_n \rightarrow S_n$ **od**

□

The statements *IF* and *DO* are defined as representatives of the general notion of an **if** or **do** statement.

$IF \mathrel{\widehat{=}}$ **if** $B_1 \rightarrow S_1 \,[]\, B_2 \rightarrow S_2$ **fi**
$DO \mathrel{\widehat{=}}$ **do** $B \rightarrow S$ **od**

A *sequential program*, also called a *component*, is just a statement. A *concurrent program*, also called a *multiprogram*, is a collection of components, together with a precondition that defines its initial states. In this paper, we will refer to a concurrent program as a program and to a sequential program as a component. The values of the variables in a program define its current data state. A variable of a component may be *local* to that component, meaning it is not read or written by any other component, or *private*, meaning it is not written by any other component, or *shared*, meaning it is written by some other component.

A component is executed by executing its atomic actions. An atomic action is an execution step that results in a single update of the control state of the whole *program*, i.e., when an atomic action is executed, the control state of the component in which the action occurs changes once, and the control state of all other components remains the same. Note that an atomic action is guaranteed to terminate when it is executed. We adopt a programming model in which an atomic action corresponds to an assignment statement, to a *skip* statement, to a guard evaluation step in an **if** or **do** statement, or to a coarse-grained atomic statement. The latter is defined by applying the 'atomicity operator' $\langle S \rangle$ to an arbitrary statement S, where the operator eliminates any control points in S so that $\langle S \rangle$ is executed atomically as just described. Note that execution of $\langle S \rangle$ is only enabled (not blocked) if execution of S is guaranteed to terminate. While this creates an impossible difficulty for the implementor, since a machine can not, in general, decide whether a statement will terminate, the use of coarse-grained atomic statements in our language allows us to nicely capture otherwise informal concepts (see [GD05]). [AO91] solve this problem syntactically, by disallowing S to contain a loop or a synchronisation statement, whereas our approach is to make it the responsibility of the programmer to ensure that a coarse-grained atomic statement is guaranteed to terminate.

Condition synchronisation in the model is achieved using the **if** statement. Execution of the guard evaluation action of an **if** statement is blocked when the guard evaluation action is not enabled, which is when all of the guards are evaluated false. A guard evaluation action of an **if** statement is therefore a conditional atomic action because it may not always be enabled. A guard evaluation action of a **do** statement is an unconditional atomic action because it is always enabled, as are *skip* and assignment actions. The programming model prescribes that on termination of an atomic action, an atomic action that follows it, if there is one, is eventually executed if it is continually enabled. This means that in the concurrent execution of a number of components, the execution of the next (continually enabled) atomic action of no component is delayed indefinitely.

2.2 Hoare triples, the *wlp* and the core theory of Owicki and Gries

If P and Q are any two predicates, and S is a statement, a *Hoare-triple*, $\{P\}\ S\ \{Q\}$ is true iff each terminating execution of S that starts in an initial state satisfying P is guaranteed to end in a final state satisfying Q. P is called the *precondition* of S and Q the *postcondition*. A predicate that appears in a Hoare-triple is also called an *assertion* and programs that have such assertions are referred to as being *annotated*. The annotation of a program also defines the program's initial state with a precondition, which is referred to as *Pre*.

Definition (*Weakest Liberal Precondition*) The *weakest liberal precondition* (*wlp*) [Dij76] predicate transformer is defined inductively as follows, where $P[\overline{x{:=}\ E}]$ denotes the textual substitution of each E_i for free occurrences of x_i in P.

1. $wlp.skip.P \equiv P$
2. $wlp.(\overline{x{:=}\ E}).P \equiv P[\overline{x{:=}\ E}]$
3. $wlp.\langle S\rangle.P \equiv wlp.S.P$
4. $wlp.(S_1;\ S_2).P \equiv wlp.S_1.(wlp.S_2.P)$
5. $wlp.IF.P \equiv (B_1 \Rightarrow wlp.S_1.P) \wedge (B_2 \Rightarrow wlp.S_2.P)$
6. *wlp* in the case of statement *DO* need no longer be first order definable [Gum99], as we do not know if or when the loop terminates. The *wlp* of a *DO* loop is defined in terms of a countable sequence of conditionals of the form $D \mathrel{\widehat{=}}$ **if** $B \rightarrow S\ [\!]\ \neg B \rightarrow skip$ **fi**.

   $$wlp.DO.P \equiv \bigvee_{n=1}^{\infty} wlp.D_n.P$$

   where D_n is the n-fold iteration of statement D.

□

The fundamental relation between Hoare-triples and *wlp* is that, for any statement S and predicates P and Q[1],

$$\{P\}\ S\ \{Q\} \equiv P \Rightarrow wlp.S.Q$$

In a program design setting it is usually most convenient to present proofs using the predicate transformer *wlp*. However, this is not always the case due to the awkwardness of the definition of *wlp* for statement *DO*, where it is more convenient to make use of the following theorem

$$\{P\}\ DO\ \{Q\} \Leftarrow ((P \wedge B \Rightarrow wlp.S.P) \wedge (P \wedge \neg B \Rightarrow Q))$$

Any predicate P that satisfies this relation is referred to as a *loop invariant*, and proving correctness of an annotated *DO* statement amounts to finding a P that satisfies this relation.

We are now in a position to state the core theory of Owicki and Gries, which defines the conditions under which a program annotation is correct.

Rule *(Local Correctness)* An assertion P in a component is *locally correct* (LC) when,

1. if P is textually preceded by program precondition *Pre*, then $Pre \Rightarrow P$
2. if P is textually preceded by $\{Q\}\ S$, then $\{Q\}\ S\ \{P\}$ holds, i.e., $Q \Rightarrow wlp.S.P$.

♦

[1] It is common to relate Hoare-triples to the total correctness predicate transformer *wp*, however, this is ill-suited to a programming paradigm in which termination is not always desired.

Rule *(Global Correctness)* An assertion P in a component is *globally correct* (GC) if for each $\{Q\}\ S$ from a different component, $\{P \wedge Q\}\ S\ \{P\}$ holds, i.e., $P \wedge Q \Rightarrow wlp.S.P$

♦

An assertion is *correct* if it is both locally and globally correct. An annotation is *correct* if all assertions are correct.

Rule *(Postcondition)* A predicate P is a valid postcondition of a program if the conjunction of the correct postconditions of the components implies P.

♦

It is useful at this point to introduce a simple example of how the theory can be used to prove a safety requirement which will serve to make the foregoing discussion concrete. Consider this program of two components A and B

Program (1) *Pre*: $x = 0$	
Component A: $x := x + 1$	Component B: $x := x + 2$

Safety: Program (1) has terminated $\Rightarrow x = 3$

PROOF that Program (1) satisfies *Safety*.

A and B are annotated locally correct (LC) and note that both satisfy part (1) of the LC rule

Program (1) *Pre*: $x = 0$	
Component A: $\{x = 0\}$ $x := x + 1$ $\{x = 1\}$	Component B: $\{x = 0\}$ $x := x + 2$ $\{x = 2\}$

Global correctness (GC) of the annotation is now arranged by weakening all four assertions, noting that this maintains LC

Program (1) *Pre*: $x = 0$	
Component A: $\{P\colon\ x = 0 \vee x = 2\}$ $x := x + 1$ $\{Q\colon\ x = 1 \vee x = 3\}$	Component B: $\{x = 0 \vee x = 1\}$ $x := x + 2$ $\{x = 2 \vee x = 3\}$

The GC of the assertions P and Q in A are calculated

$wlp.(x := x + 2).P$
$\equiv$ {Substituting the value of P}

$$
\begin{array}{ll}
 & wlp.(x{:=}\, x+2).(x=0 \vee x=2) \\
\Leftarrow & \quad \{\text{By definition of } wlp\} \\
 & x=0 \\
\equiv & \quad \{\text{By logic}\} \\
 & (x=0 \vee x=2) \wedge (x=0 \vee x=1)
\end{array}
$$

and

$$
\begin{array}{ll}
 & wlp.(x{:=}\, x+2).Q \\
\equiv & \quad \{\text{Substituting the value of } Q\} \\
 & wlp.(x{:=}\, x+2).(x=1 \vee x=3) \\
\Leftarrow & \quad \{\text{By definition of } wlp\} \\
 & x=1 \\
\equiv & \quad \{\text{By logic}\} \\
 & (x=1 \vee x=3) \wedge (x=0 \vee x=1)
\end{array}
$$

Finally, the conjunction of the two final assertions of A and B establishes the desired safety requirement

$$(x=1 \vee x=3) \wedge (x=2 \vee x=3) \quad \equiv \quad x=3$$ ■

The simplicity of the core theory is reflected in its limited power. The lack of a means to reason about a program's control state means that safety requirements that are clearly met may not be provable, such as in the following program.

Program (2) *Pre*: $x=0$	
Component A: $x{:=}\, x+1$	Component B: $x{:=}\, x+1$

Safety: Program (2) has terminated $\Rightarrow x=2$

It is an interesting exercise to convince yourself that this safety requirement is not provable in the core theory. The solution in [OG76] is to add auxiliary information into a program which could be used in its correctness proof. We start by defining an *auxiliary assignment*, which is an assignment to a fresh variable called an *auxiliary variable* and is different from all program variables. The assignment may only appear as part of an atomic action, hence, does not introduce any new control points. We require that actions remain well-formed when all auxiliary assignments are removed. Furthermore, as addition of auxiliary information should not affect control and data states of the original program, auxiliary variables may not appear in any guard and assigned to a non-auxiliary variable.

Returning to the example of Program (2), we augment the program with auxiliary assignments to fresh variables $pc.A$ and $pc.B$ to give us the following program.

Program (3) *Pre*: $x=pc.A=pc.B=0$	
Component A: $x{:=}\, x+1 \parallel pc.A{:=}\, 1$	Component B: $x{:=}\, x+1 \parallel pc.B{:=}\, 1$

Safety: Program (3) has terminated $\Rightarrow x=2$

PROOF that Program (3) satisfies *Safety*.

This is now much as for Program (1). The two components can be annotated for LC

Program (3) *Pre*: $x = pc.A = pc.B = 0$	
Component A: $\{x = 0\}\{pc.A = 0\}$ $x := x + 1 \parallel pc.A := 1$ $\{x = 1\}\{pc.A = 1\}$	Component B: $\{x = 0\}\{pc.B = 0\}$ $x := x + 1 \parallel pc.B := 1$ $\{x = 1\}\{pc.B = 1\}$

GC is arranged by a combination of strengthening and weakening these assertions

Program (3) *Pre*: $x = pc.A = pc.B = 0$	
Component A: P: $\{(x = 0 \wedge pc.B = 0) \vee (x = 1 \wedge pc.B = 1)\}$ $\{pc.A = 0\}$ $x := x + 1 \parallel pc.A := 1$ Q: $\{(x = 1 \wedge pc.B = 0) \vee (x = 2 \wedge pc.B = 1)\}$ $\{pc.A = 1\}$	Component B: $\{(x = 0 \wedge pc.A = 0) \vee (x = 1 \wedge pc.A = 1)\}$ $\{pc.B = 0\}$ $x := x + 1 \parallel pc.B := 1$ $\{(x = 1 \wedge pc.A = 0) \vee (x = 2 \wedge pc.A = 1)\}$ $\{pc.B = 1\}$

As before, the GC of P and Q in A are calculated

$$
\begin{array}{ll}
 & wlp.(x := x + 1 \parallel pc.B := 1).(((x = 0 \wedge pc.B = 0) \vee (x = 1 \wedge pc.B = 1)) \wedge pc.A = 0) \\
\equiv & \quad \{\text{By definition of } wlp\} \\
 & x = 0 \wedge pc.A = 0 \\
\Leftarrow & \quad \{\text{By logic}\} \\
 & ((x = 0 \wedge pc.B = 0) \vee (x = 1 \wedge pc.B = 1)) \wedge pc.A = 0 \wedge pc.B = 0
\end{array}
$$

and

$$
\begin{array}{ll}
 & wlp.(x := x + 1 \parallel pc.B := 1).(((x = 1 \wedge pc.B = 0) \vee (x = 2 \wedge pc.B = 1)) \wedge pc.A = 1) \\
\equiv & \quad \{\text{By definition of } wlp\} \\
 & x = 1 \wedge pc.A = 1 \\
\Leftarrow & \quad \{\text{By logic}\} \\
 & ((x = 1 \wedge pc.B = 0) \vee (x = 2 \wedge pc.B = 1)) \wedge pc.A = 1 \wedge pc.B = 0
\end{array}
$$

Finally, the conjunction of the two final assertions of A and B establishes the desired safety requirement.

$$((x = 1 \wedge pc.B = 0) \vee (x = 2 \wedge pc.B = 1)) \wedge pc.B = 1 \quad \Rightarrow \quad x = 2$$ ■

Noting that Program (3) is just Program (2) with auxiliary assignments to $pc.A$ and $pc.B$ superimposed on it, we are entitled to conclude that Program (2) satisfies the same safety requirement, because the two programs are equivalent in having the same data and control states.

3 The extended theory of Owicki and Gries

It is fairly clear, so far as reasoning about progress is concerned, that the theory of Owicki and Gries is deficient because it lacks a systematic means to describe a program's control state. Any extension to the theory must therefore provide for this, and the extension to be described in this section has two parts. First, control points in a component are named by naming the atomic action to be executed at the corresponding point. This is done by labelling all of the atomic actions in the component. Second, an auxiliary variable is introduced into each component in a way that models its 'program counter', i.e., the value of this variable indicates the active control point in the component, which is just the label of the atomic action that corresponds to that control point.

Sections 3.1 and 3.2 introduces the twin notions of a labelled program and a program counter while Section 3.3 reviews the reasons why program counters were chosen over control predicates as the formalisation of program control states.

3.1 Labelled statements

The first step toward describing an active control point in a statement requires being able to refer to the next atomic action to be executed. We do this by assigning a unique label to each atomic action that occurs in the statement. The label of a statement's initial atomic action will be called the *initial label* of that statement. In addition, a label will be assigned to the end of the statement which will be called the *final label* of the statement. A final label of a statement will always label the initial atomic action of a statement that follows it. However, if there is no following statement, then the final label does not refer to any atomic action, but simply marks the end of the statement.

Definition *(Labelled Statement)*

1. A labelled *skip* statement has the form i: *skip* j: where i is the initial label and j is the final label.
2. A labelled assignment statement has the form $i: x := E\ j$: where i is the initial label and j is the final label.
3. A labelled sequential statement has the form $i: S_1;\ j: S_2\ k$: where i is the initial label of statement S_1, j is the final label of S_1, j is the initial label of statement S_2 and k is the final label of S_2.
4. A labelled coarse-grained atomic statement $\langle S \rangle$ has the form $i: \langle S \rangle\ j$: where i is the initial label and j is the final label, and statement S is not labelled.
5. A labelled statement IF has the form

   $$i: \textbf{if}\ B_0 \rightarrow\ j: S_0\ [\!]\ B_1 \rightarrow\ k: S_1\ \textbf{fi}\ l:$$

   where i is the initial label of IF and l is the final label of IF, i is the label of the initial atomic action of IF, which is the guard evaluation action, and j and k are the final labels of the guard evaluation action. j is the initial label of statement S_0, k is the initial label of statement S_1 and l is the final label of both S_0 and S_1.
6. A labelled statement DO has the form

   $$i: \textbf{do}\ B \rightarrow\ j: S\ \textbf{od}\ k:$$

   where i is the initial label of DO and k is the final label of DO, i is the label of the initial atomic action of DO, which is the guard evaluation action, and j and k are final labels of the guard evaluation action. j is the initial label of statement S and i is the final label of S.
7. If i and j are the initial labels for two different actions of any statement, then $i \neq j$

□

In what follows $A.i$ will be used to denote 'the atomic action in component A labelled i' whenever i is not the final label of component A.

3.2 Modelling program counters

There are essentially two ways of using this additional information that labelled statements provide. One way is to introduce into the logic new control predicates to express propositions such as, for instance, that 'control in component A is at the atomic action labelled i'. This kind of approach is taken in ([Sch97], pp96-108, pp136-140) and in ([Lam87]), but the cost is that the familiar axioms of Hoare logic, as presented in Section 2.2, must be given up in favour of generalised axioms that take account of the fact that, say, $\{P\}$ *skip* $\{P\}$ is no longer true for all P (for example, when P asserts that 'control is at the *skip* action'). A further cost is that new axioms must be introduced to capture the intended interpretation of the new control predicates. The desire to make only conservative extension to the theory of Owicki and Gries, prompted by the desire to retain old, familiar and trusted ways (the *wlp*), has led us to resist this approach in favour of the use of auxiliary variables to reason about the control state.

Consequently, we formalise a program's control state in the following way. An auxiliary variable is introduced into each component in a way that models its 'program counter', i.e., the value of this variable indicates the active control point in the component, which is just the label of the next atomic action to be executed, or the end of the component if no such action exists. Given a component A, this variable $pc.A$ will be called the *program counter* of A, and its essence is to record the control state of A, but in so doing to change neither the program's control state nor its data state. Given this essence, program counter $pc.A$ must be updated at every atomic action in A in a way that assigns $pc.A$ a final label of that action. This is done by superimposing an auxiliary assignment to $pc.A$ onto every atomic action in A in the following way.

Definition (*Program Counter*) Given a program with precondition *Pre* and labelled component A, variable $pc.A$ is the *program counter* of A when

1. $pc.A$ is a local variable of A.
2. If i is the initial label of A then $Pre \Rightarrow pc.A = i$
3. A labelled *skip* statement has the form $i{:}\,\langle skip;\ pc.A{:=}\ j\rangle\ j{:}\ .$
4. A labelled assignment statement has the form $i{:}\,x{:=}\ E \parallel pc.A{:=}\ j\ j{:}\ .$
5. A labelled coarse-grained atomic statement has the form $i{:}\,\langle S;\ pc.A{:=}\ j\rangle\ j{:}\ .$
6. A labelled statement IF has the form

   $$i{:}\,\mathbf{if}\ \langle B_1 \rightarrow\ pc.A{:=}\ j\rangle\ j{:}\,S_1\ [\!]\ \langle B_2 \rightarrow\ pc.A{:=}\ k\rangle\ k{:}\,S_2\ \mathbf{fi}\ l{:}$$

7. A labelled statement DO has the form

   $$i{:}\,\mathbf{do}\ \langle B \rightarrow\ pc.A{:=}\ j\rangle\ j{:}\,S\ \ [\!]\ \langle \neg B \rightarrow pc.A{:=}\ k\rangle\ \mathbf{od}\ k{:}$$

Given that guard evaluation is an atomic action (as it changes the program control state whenever a guard is evaluated *true*), and given that a program counter must be updated at *every* atomic action in a component, we are required to extend the grammar of statements IF and DO in order to make explicit the state change that can accompany a guard evaluation. To this end we modify the syntax of guarded command $B \rightarrow j{:}\,S$ to $\langle B \rightarrow pc.A{:=}\ j\rangle\ j{:}\,S$ so that the transfer of program control from the guard evaluation to the initial action of S when guard B evaluates *true* is made explicit. Note how atomicity brackets $\langle\ \rangle$ are used to indicate that the update of the program counter is part of the guard evaluation. However, we acknowledge that this grammar is awkward, because it is semantically misleading whenever a statement consists of several alternatives. For example, in statement

$$i{:}\,\mathbf{if}\ \langle B_1 \rightarrow\ pc.A{:=}\ j\rangle\ j{:}\,S_1\ [\!]\ \langle B_2 \rightarrow\ pc.A{:=}\ k\rangle\ k{:}\,S_2\ \mathbf{fi}\ l{:}$$

the two pairs of atomicity brackets suggest two atomic guard evaluations, which is not the case, rather there is one atomic guard evaluation labelled by i, which has three outcomes, the first where guard B_1 is

evaluated to *true* and control passes to the initial action of S_1 labelled by j, the second where guard B_2 is evaluated to *true* and control passes to the initial action of S_2 labelled by k, and the third where both guards B_1 and B_2 are evaluated to *false* and control remains at the guard evaluation action labelled by i.

The case of statement DO

$$i{:}\,\mathbf{do}\ B \rightarrow\ j{:}\,S\ \mathbf{od}\ k{:}$$

is further complicated by the fact that the loop is not a blocking statement, which is to say that when guard B is evaluated *false* control does not remain at the guard evaluation labelled by i, but rather it passes to the control point labelled by k. This transfer of control requires an explicit update to the program counter, which we have accommodated by changing the grammar of the DO statement in a way that makes this outcome of the guard evaluation explicit

$$i{:}\,\mathbf{do}\ B \rightarrow\ j{:}\,S\ \ [\!]\ \neg B \rightarrow\ \mathbf{od}\ k{:}$$

The DO statement now admits a guarded command $\neg B \rightarrow$ with an empty command, which, if selected, has the total effect on the program state of passing control to the control point labelled by k. The operational semantics of this syntactically modified DO statement is unchanged, with the sole purpose of the modification being to introduce a peg on which to hang the assignment $pc.A{:=}\ k$.

Finally, note that we are free to interpret predicate $pc.A = i$ to mean that 'control in A is at $A.i$' because $pc.A = i$ is a correct precondition of $A.i$ and because labels are unique. LC follows from the definition of $pc.A$, and GC follows from the same, on account of $pc.A$ being a local variable of A.

3.3 Program counters vs. control predicates

Recalling that the reason for choosing program counters over control predicates has been driven by a desire to make only conservative changes to the theory of Owicki and Gries, we can view this choice as one of a superficial (i.e., syntactic) change to guarded commands in order to make explicit the way that a guard evaluation can change the control state, over a significant (i.e., semantic) change to the program logic. The chief practical gains are that we are able to retain the semantics of *wlp* as the logical basis of the programming model and that the absence of primitive control predicates means that we do not need to introduce additional logical rules to define them. The core theory of Owicki and Gries as described in Section 2.2 therefore remains the same under the changes described in Sections 3.1 and 3.2, and the definition of the *wlp* predicate transformer is extended to a labelled statement with program counter pc as follows

1. $wlp.(i{:}\,\langle skip;\ pc{:=}\ j\rangle\ j{:}\,).P \equiv wlp.(pc{:=}\ j).P$
2. $wlp.(i{:}\,(\overline{x{:=}\ E}\ \|\ pc{:=}\ j)\ j{:}\,).P \equiv P[\overline{x{:=}\ E}\ \|\ pc{:=}\ j]$
3. $wlp.(i{:}\,\langle S;\ pc{:=}\ j\rangle\ j{:}\,).P \equiv wlp.(S;\ pc{:=}\ j).P$
4. $wlp.(i{:}\,S_1;\ j{:}\ \ S_2\ k{:}\,).P \equiv wlp.(i{:}\,S_1\ j{:}\,).(wlp.(j{:}\,S_2\ k{:}\,).P)$
5. $wlp.(i{:}\,\mathbf{if}\ \langle B_1 \rightarrow\ pc{:=}\ j\rangle\ j{:}\,S_1\ [\!]\ \langle B_2 \rightarrow\ pc{:=}\ k\rangle\ k{:}\,S_2\ \mathbf{fi}\ l{:}\,).P$
   $\equiv$
   $(B_1 \Rightarrow wlp.(pc{:=}\ j).(wlp.S_1.P)) \wedge (B_2 \Rightarrow wlp.(pc{:=}\ k).(wlp.S_2.P))$
6. $\{P\}\ i{:}\,\mathbf{do}\ \langle B \rightarrow\ pc{:=}\ j\rangle\ j{:}\,S\ \ [\!]\ \langle \neg B \rightarrow pc{:=}\ k\rangle\ \mathbf{od}\ k{:}\ \{Q\}$
   $\Leftarrow$
   $(P \wedge B \Rightarrow wlp.(pc{:=}\ j).(wlp.S.P)) \wedge (P \wedge \neg B \Rightarrow wlp.(pc{:=}\ k).Q)$

It is noteworthy that typical axioms [Lam87, AS89] that are required to define the meaning of a control predicate now become easy derived rules of the program counters model.

1. Each component has at most one active control point. This is trivial on account of $(\forall\, i, j\ :j \neq i:\ pc.A = i \Rightarrow pc.A \neq j)$ and the uniqueness of labels.
2. Each component has at least one active control point. This holds on account of the invariance of $(\exists\, i ::\ pc.A = i)$.
3. Execution of an atomic statement in component $B \neq A$ does not change the active control point in component A. This is trivial on account of A's program counter being a local variable of A.

Against the advantages of using program counters, the chief drawback is the syntactic complexity that the program counter assignments add to the program under consideration. However, this added complexity is nicely avoided in practice by making the assignments implicit in the program. In effect, this amounts to redefining the *wlp* for a labelled statement with implicit program counter *pc* as follows

1. $wlp.(i{:}\ skip\ j{:}).P \equiv wlp.(pc{:=}\ j).P$
2. $wlp.(i{:}\ \overline{x{:=}\ E}\ j{:}).P \equiv P[\overline{x{:=}\ E} \parallel pc{:=}\ j]$
3. $wlp.(i{:}\ \langle S\rangle\ j{:}).P \equiv wlp.(S;\ pc{:=}\ j).P$
4. $wlp.(i{:}\ S_1;\ j{:}\ S_2\ k{:}).P \equiv wlp.(i{:}\ S_1\ j{:}).(wlp.(j{:}\ S_2\ k{:}).P)$
5. $wlp.(i{:}\ \mathbf{if}\ B_1 \rightarrow\ j{:}\ S_1 \,[]\, B_2 \rightarrow\ k{:}\ S_2\ \mathbf{fi}\ l{:}).P$
   $\equiv$
   $(B_1 \Rightarrow wlp.(pc{:=}\ j).(wlp.S_1.P)) \wedge (B_2 \Rightarrow wlp.(pc{:=}\ k).(wlp.S_2.P))$
6. $\{P\}\ i{:}\ \mathbf{do}\ B \rightarrow\ j{:}\ S\ \mathbf{od}\ k{:}\ \{Q\}$
   $\Leftarrow$
   $(P \wedge B \Rightarrow wlp.(pc{:=}\ j).(wlp.S.P)) \wedge (P \wedge \neg B \Rightarrow wlp.(pc{:=}\ k).Q)$

and this is what we do.

4 A logic of progress for the extended theory

As we now have the means to reason about the control state of a program, we are now in a position to extend the theory to support reasoning about progress requirements. The rules for progress in the extended theory are described in Section 4.1. Section 4.2 describes an application of the new logic to a program design task, which compares favourably to the treatment in ([FvG99], pp2 07-212) and Section 4.3 describes a second application of the logic, this time to the proof of correctness of a program transformation.

4.1 Rules of progress

As already remarked in Section 1, the logic to be presented is almost just that of UNITY ([CM88], pp47-74), where the notion of progress is formalised using the relation *leads-to* (denoted $\rightsquigarrow$), where, for any predicates P and Q, $P \rightsquigarrow Q$ holds if it is always the case that in a program state in which P holds, execution of the program is such that a program state will eventually be reached in which Q holds. In temporal logic terms, $P \rightsquigarrow Q \equiv \Box(P \Rightarrow \Diamond Q)$. In order to axiomatize this relation, we begin by defining the notion of *unless* (**un**).

Definition If P and Q are any two predicates, P **un** Q holds if

$$\{P \wedge \neg Q \wedge U\}\ S\ \{P \vee Q\}$$

holds for all atomic statements $\{U\}\ S$, where U denotes the precondition of S in the annotated program. □

Relation *unless* says that a program state in which P holds and Q does not, is perpetuated until a state is reached in which Q holds. But note that this does not guarantee that Q will ever hold, for (an extreme) example, *true* **un** Q holds for all Q, including *false*. To formalise progress properties we also need a notion of what it means for a statement to establish a predicate given that it is not yet true. In ([CM88], pp50-52) this is formalised by the relation *ensures*, which forms the basis of the definition of *leads-to*. In our setting, and purely for presentational reasons, we have chosen not to define *ensures*, but rather to define the basic part of *leads-to* directly in terms of the several forms of atomic action in the programming language. More substantially, the basic part of our definition of *leads-to*, which is the point at which the relation is bound to the program under consideration, is the only point at which the two definitions of *leads-to* differ, the inductive part of our definition being identical to that in ([CM88], p52). However, this difference in the basic definition of *leads-to* is an essential difference, on account of the fundamentally different programming model that is used here and in UNITY.

We remind ourselves that for the basic part of the definition of *leads-to*, the atomic actions are *skip*, assignment, guard evaluation and coarse-grained atomic statements of the form $\langle S \rangle$ for arbitrary statement S. A judgment $P \rightsquigarrow Q$ arrived at using this rule ensures that if a program state is reached in which P holds, execution of the program is such that P will continue to hold until a program state is reached in which Q holds, and, further, a state in which Q holds will be reached. We will call this the 'immediate progress' rule because it allows us to actually exhibit an atomic action that is guaranteed to bring Q about.

Rule *(Immediate Progress Rule)* $P \rightsquigarrow Q$ holds whenever there is a labelled statement with initial label i in a component with program counter pc and

1. P **un** Q
2. $P \wedge \neg Q \Rightarrow pc = i$
3. (a) The statement is an assignment or *skip* statement $i\colon S\ j\colon$ and,
   $$P \wedge \neg Q \Rightarrow wlp.(i\colon S\ j\colon).Q$$
   (b) The statement is an *IF* statement $i\colon \mathbf{if}\ B_0 \rightarrow\ j\colon S_0\ [\!]\ B_1 \rightarrow\ k\colon S_1\ \mathbf{fi}\ l\colon$ and,
   i. $P \wedge \neg Q \Rightarrow B_0 \vee B_1$
   ii. $(P \wedge \neg Q \wedge B_0 \Rightarrow wlp.(pc{:=}\ j).Q)) \wedge$
   $(P \wedge \neg Q \wedge B_1 \Rightarrow wlp.(pc{:=}\ k).Q)$
   (c) The statement is a *DO* statement $i\colon \mathbf{do}\ B \rightarrow\ j\colon S\ \mathbf{od}\ k\colon$ and,
   $(P \wedge \neg Q \wedge B \Rightarrow wlp.(pc{:=}\ j).Q) \wedge$
   $(P \wedge \neg Q \wedge \neg B \Rightarrow wlp.(pc{:=}\ k).Q)$
   (d) The statement is a coarse-grained atomic statement $i\colon \langle S \rangle\ j\colon$
   $$P \wedge \neg Q \Rightarrow wp.S.(wlp.(pc{:=}\ j).Q)$$

♦

To make sense of this rule we provide these interpretative notes. $P \rightsquigarrow Q$ is here justified on the basis of being able to actually exhibit a continually enabled atomic action at an active control point that makes Q true when it is executed. To see how the rule formalises this, we first note that $P \wedge \neg Q$ is assumed. Clause 1 of the rule establishes that P remains true as long as $\neg Q$ is true. Clause 2 establishes that control is at an atomic action labelled i in a component. Clause 3 establishes that this action is enabled when $P \wedge \neg Q$ is true, and that its execution makes Q true. It follows from clause 1 that the action is continually enabled as long as $\neg Q$ is true. It then follows from the programming model that the action is eventually executed. Clause 3 is separated into four cases to cover the three kinds of atomic action. In case (3a), an assignment action is always enabled and it is enough to ensure that its execution makes Q true. In case (3b), a guard evaluation action in an **if** statement is not always enabled and so clause (3bi) ensures that it is enabled when $P \wedge \neg Q$ is true. Clause (3bii) further ensures that its execution makes Q true. In case (3c), a guard evaluation action in a **do** statement is always enabled and it is again enough to ensure that its execution makes Q true. In case (3d), as it is possible for a coarse-grained

atomic action to be disabled, we use of the predicate transformer *wp* to ensure that the action is enabled when $P \wedge \neg Q$ is true. Further, execution of S must make $Q[pc{:=}\, j]$ true.

The inductive part of the definition of *leads-to* is given by

Rule *(Inductive Progress Rules)*

(Transitivity) $P \rightsquigarrow R \Leftarrow P \rightsquigarrow Q \wedge Q \rightsquigarrow R$

(Disjunction) For any set W, $(\exists\, i : i \in W\colon\ P_i) \rightsquigarrow Q \Leftarrow (\forall\, i : i \in W\colon\ P_i \rightsquigarrow Q)$

♦

The rule of transitivity requires no explanation. The rule of disjunction, in its finite application of, say, two progress assertions, amounts to the inference that if $P_0 \rightsquigarrow Q$ and $P_1 \rightsquigarrow Q$ then $P_0 \vee P_1 \rightsquigarrow Q$. [CM88] also present a thorough treatment of a collection of derived rules for the relation, all of which remain true in our setting, and which are listed below. The proofs of these derived rules are presented in Appendix A.

Rule *(Derived Progress Rules)*

1. $P \rightsquigarrow Q \Leftarrow (P \Rightarrow Q)$ (Implication Theorem)
2. $\neg P \Leftarrow (P \rightsquigarrow \mathit{false})$ (Impossibility Theorem)
3. $((\exists\, m : m \in W\colon\ P.m) \rightsquigarrow (\exists\, m : m \in W\colon\ Q.m))$
   $\Leftarrow (\forall\, m : m \in W\colon\ P.m \rightsquigarrow Q.m)$ (Disjunction Theorem)
4. $(P \rightsquigarrow Q \vee R) \Leftarrow (P \rightsquigarrow Q \vee B) \wedge (B \rightsquigarrow R)$ (Cancellation Theorem)
5. $(P \wedge R \rightsquigarrow (Q \wedge R) \vee B) \Leftarrow (P \rightsquigarrow Q) \wedge (R \textbf{ un } B)$ (PSP (Progress-Safety-Progress) Theorem)
6. Let M be a total function from program states to set W. Let $(W, <)$ be well-founded. Variable m in the following premiss ranges over W and predicates P and Q do not contain free occurrences of variable m. Then,
   $(\forall\, m ::\ P \wedge M = m \rightsquigarrow (P \wedge M < m) \vee Q) \Rightarrow (P \rightsquigarrow Q)$
   (Induction Theorem)
7. Let $P.i$ and $Q.i$ be predicates where i ranges over a finite set. Then,
   $(\forall\, i ::\ (P.i \rightsquigarrow Q.i \vee B) \wedge (Q.i \textbf{ un } B)) \Rightarrow ((\forall\, i ::\ P.i) \rightsquigarrow (\forall\, i ::\ Q.i) \vee B)$
   (Completion Theorem)

♦

The remainder of this section gives two examples of how the new logic can be used. The first presents an application of the logic to a program design task, which compares favourably to the treatment in ([FvG99], pp207-212), and the second presents a proof of correctness of a program transformation called the "guard conjunction lemma", which is taken from the same source ([FvG99], pp118-120).

4.2 The initialisation protocol

The first example is taken from [FvG99] where it appears as both an exercise in verification (p84) and as an exercise in design (p207). Here we present an alternative design that starts with the following program

The Initialisation Protocol *Pre*: *true*	
Component X: $Init.X$; $y{:=}\, \mathit{false}$; $\langle$**if** $y \rightarrow$ *skip* **fi**$\rangle$; $S.X$	Component Y: $Init.Y$; $x{:=}\, \mathit{false}$; $\langle$**if** $x \rightarrow$ *skip* **fi**$\rangle$; $S.Y$

Progress: There is no individual deadlock

The safety requirement of the initialisation protocol is omitted from the specification on account of this program already satisfying it, the requirement being that X cannot begin execution of code $S.X$ until Y has completed execution of code $Init.Y$, and vice versa. This requirement is maintained provided that only assignment $y{:=}\ true$ in Y is allowed, and this only after $Init.Y$, which is nicely ensured by restricting attention to the protocol code

The Initialisation Protocol – simplified and labelled *Pre*: $pc.X = pc.Y = 1$	
Component X: 1: $y{:=}\ false$; 2: $\langle$**if** $y \rightarrow$ *skip* **fi**$\rangle$ 3:	Component Y: 1: $x{:=}\ false$; 2: $\langle$**if** $x \rightarrow$ *skip* **fi**$\rangle$ 3:

Progress: $pc.X = 2 \leadsto pc.X = 3$
Topology: Only $y{:=}\ true$ is allowed in Y

A canonical pattern for ensuring progress at a guarded skip is to prove that 'waiting at the guard leads to the guard becoming true' and 'waiting at the guard when the guard is true leads to being after the guard'. Thus, *Progress* is proved by hypothesising (1).

$$
\begin{array}{ll}
 & pc.X = 2 \\
\leadsto & \quad \{\text{By hypothesis (1)}\} \\
 & pc.X = 2 \wedge y \\
\leadsto & \quad \{IF \text{ progress rule with } X.2,\ y \text{ is GC in } X\} \\
 & pc.X = 3
\end{array}
$$

A canonical pattern for ensuring that 'waiting at a guard leads to the guard becoming true' is to show the the rest of the program leads to the guard becoming true. In this case the rest of the program is just component Y, so, (1) is proved at the cost of (2)

$$
\begin{array}{ll}
 & pc.X = 2 \leadsto pc.X = 2 \wedge y \\
\Leftarrow & \quad \{\text{Derived rule}\} \\
 & (\forall\, i ::\ pc.X = 2 \wedge \neg y \wedge pc.Y = i \leadsto pc.X = 2 \wedge y) \quad (2)
\end{array}
$$

(2) is proved by case analysis for $pc.Y \in \{1, 2, 3\}$

$$
\begin{array}{ll}
 & pc.X = 2 \wedge \neg y \wedge pc.Y = 1 \leadsto pc.X = 2 \wedge y \quad (3) \\
 & pc.X = 2 \wedge \neg y \wedge pc.Y = 2 \leadsto pc.X = 2 \wedge y \quad (4) \\
 & pc.X = 2 \wedge \neg y \wedge pc.Y = 3 \leadsto pc.X = 2 \wedge y \quad (5)
\end{array}
$$

For (3), on account of $Y.1$ not hampering progress, on account of being an assignment, and being orthogonal to $pc.X = 2 \wedge \neg y$, we opt for deferring the obligation to make y true, by delegating the task to $Y.2$. (3) is therefore proved

$$
\begin{array}{ll}
 & pc.X = 2 \wedge \neg y \wedge pc.Y = 1 \\
\leadsto & \quad \{\text{Assignment progress rule with } Y.1\} \\
 & pc.X = 2 \wedge \neg y \wedge pc.Y = 2 \\
\leadsto & \quad \{\text{By (4)}\} \\
 & pc.X = 2 \wedge y
\end{array}
$$

For (4), since $Y.2$ is a guarded skip, deadlock is avoided by requiring invariance of

$$
\begin{array}{ll}
 & pc.X = 2 \wedge \neg y \wedge pc.Y = 2 \wedge \neg x \Rightarrow \mathit{false} \qquad (I) \\
\equiv & \quad \{\text{By logic}\} \\
 & pc.Y = 2 \Rightarrow pc.X \neq 2 \vee y \vee x
\end{array}
$$

Since $pc.X$ is local to X and because of the topological constraint on Y, there is no choice but to introduce assignment (y:= *true*) at $Y.2$

The Initialisation Protocol – refinement 1 *Pre*: $pc.X = pc.Y = 1$	
Component X: 1: y:= *false*; 4: x:= *true*; 2: $\langle$**if** $y \rightarrow$ *skip* **fi**$\rangle$ 3:	Component Y: 1: x:= *false*; 4: y:= *true*; $\{pc.X \neq 2 \vee y \vee x\}$ 2: $\langle$**if** $x \rightarrow$ *skip* **fi**$\rangle$ 3:

(4) is now proved as follows

$$
\begin{array}{ll}
 & pc.X = 2 \wedge \neg y \wedge pc.Y = 2 \\
\rightsquigarrow & \quad \{\text{By } (I)\} \\
 & pc.X = 2 \wedge \neg y \wedge pc.Y = 2 \wedge x \\
\rightsquigarrow & \quad \{IF \text{ progress rule with } Y.2\} \\
 & pc.X = 2 \wedge \neg y \wedge pc.Y = 3 \\
\rightsquigarrow & \quad \{\text{By (5)}\} \\
 & pc.X = 2 \wedge y
\end{array}
$$

For (5), we opt for the assignment progress rule and a second assignment (y:= *true*) at $Y.3$

The Initialisation Protocol – refinement 2 *Pre*: $pc.X = pc.Y = 1$	
Component X: 1: y:= *false*; 4: x:= *true*; 2: $\langle$**if** $y \rightarrow$ *skip* **fi**$\rangle$; 3: x:= *true* 5:	Component Y: 1: x:= *false*; 4: y:= *true*; 2: $\langle$**if** $x \rightarrow$ *skip* **fi**$\rangle$; 3: y:= *true* 5:

But note that this derivation is typical in its interplay between proof and program development, and the new code at $Y.4$ and $Y.3$ has extended the case analysis for (2) to cases $pc.Y \in \{1, 2, 3, 4, 5\}$. Case $pc.Y = 4$ is again by the assignment progress rule, but case $pc.Y = 5$ is a different matter. Evidently, the assignment progress rule is not an option here for reason of infinite regress, so we look to arrange invariance of J

$$
\begin{array}{ll}
 & \mathit{false} \\
\equiv & \quad \{J\colon\ pc.Y = 5 \Rightarrow y\} \\
 & pc.X = 2 \wedge \neg y \wedge pc.Y = 5 \\
\rightsquigarrow & \quad \{\text{Implication theorem}\} \\
 & pc.X = 2 \wedge y
\end{array}
$$

The Initialisation Protocol – annotated for progress *Pre*: $pc.X = pc.Y = 1$	
Component X:	Component Y:
1: y:= *false*;	1: x:= *false*;
4: x:= *true*;	4: y:= *true*;
2: $\langle$**if** $y \rightarrow$ *skip* **fi**$\rangle$;	2: $\langle$**if** $x \rightarrow$ *skip* **fi**$\rangle$;
3: x:= *true*	3: y:= *true*
5:	$\{y\}$
	5:

For GC of assertion y at $Y.5$ we look to strengthen J with $pc.X \neq 1$

$$pc.Y = 5 \Rightarrow y \wedge pc.X \neq 1 \qquad (J)$$

which induces the following correct annotation of Y

The Initialisation Protocol – correctly annotated *Pre*: $pc.X = pc.Y = 1$	
Component X:	Component Y:
1: y:= *false*;	1: x:= *false*;
4: x:= *true*;	$\{x \Rightarrow pc.X \neq 1\}$
2: $\langle$**if** $y \rightarrow$ *skip* **fi**$\rangle$;	4: y:= *true*;
3: x:= *true*	2: $\langle$**if** $x \rightarrow$ *skip* **fi**$\rangle$;
5:	$\{pc.X \neq 1\}$
	3: y:= *true*
	$\{y\}\{pc.X \neq 1\}$
	5:

GC of $pc.X \neq 1$ is for free because every action in X makes it true on account of $X.1$ being the initial action of X. This concludes the derivation.

The example is a nice one for two reasons. First, because the problem itself is quite delicate, as can be seen by reworking the design from the point at which it was decided to establish (3) by the transitivity rule rather than by the assignment progress rule. The alternative path leads all the way to

Pre: $pc.X = pc.Y = 1$	
Component X:	Component Y:
6: x:= *true*;	6: y:= *true*;
1: y:= *false*;	1: x:= *false*;
4: x:= *true*;	4: y:= *true*;
2: $\langle$**if** $y \rightarrow$ *skip* **fi**$\rangle$;	2: $\langle$**if** $x \rightarrow$ *skip* **fi**$\rangle$;
3: x:= *true*	3: y:= *true*
5:	$\{y\}\{?\ pc.X \neq 1\}$
	5:

but now the derivation falls down on account of the (lack of) GC of $pc.X \neq 1$ at $Y.5$.

Second, while the derivation is marked by a complete absence of operational thinking, yet it was completely driven by progress concerns. This is just what we want to see in a problem like this where progress

is of the essence. In this regard, it is instructive to compare it to the derivation in [FvG99] and to note there the authors closing remark that "we have to admit that, no matter how crisp the final solution turned out to be, its derivation seems to be driven by hope and a kind of opportunism."(p212). In our view, this is not true in the present case, rather we see this derivation as a small step toward our larger goal of developing a method of program derivation in which progress requirements are given equal consideration with safety requirements.

4.3 The guard conjunction lemma

The guard conjunction lemma ([FvG99], pp118-120) describes a correct program transformation by justifying the replacement of a guarded skip with a (coarse-grained) conjunctive guard $B \wedge C$ by a pair of (fine-grained) guarded skips with guards B and C, when B is GC in the component in which the guarded skip occurs. The lemma states that the transformation preserves the safety and progress properties of the original program, and it is noteworthy that the proof of the latter part is outside of the scope of the basic theory of Owicki and Gries (as presented in Section 2.2). Thus, we are told by Feijen and van Gasteren that the basic theory "is not suited for proving [progress]. Fortunately, Dr. J. Hooman proved it for us. He did so by considering the sets of all possible computations that can be evoked by the original and by the new system, respectively, and then showing that the two systems have the same properties as far as deadlock and individual progress are concerned. The proof is not for free and we are grateful to him for having designed it for us." ([FvG99], p118). The purpose of this section is to show how the guard conjunction lemma can be proved in the extended theory of Owicki and Gries (as presented in Section 4). The lemma states

Guard Conjunction Lemma For a globally correct B, guarded command

$$i{:}\ \langle\ \textbf{if}\ B \wedge C \rightarrow S\ \textbf{fi}\ \rangle\ j{:}$$

may be replaced by

$$i{:}\ \langle\ \textbf{if}\ B \rightarrow \mathit{skip}\ \textbf{fi}\ \rangle;\quad k{:}\ \langle\ \textbf{if}\ C \rightarrow S\ \textbf{fi}\ \rangle\ j{:}$$

without

(i) impairing the correctness of the annotation

(ii) introducing total deadlock

(iii) endangering individual progress

For the sake of completeness, we begin by reproducing the proof of (i).

Proof of (i)

$$\begin{array}{ll} & \langle\ \textbf{if}\ B \wedge C \rightarrow S\ \textbf{fi}\ \rangle \\ \sqsubseteq & \quad \{\text{adding a } \mathit{skip} \text{ is harmless}\} \\ & \langle\ \textbf{if}\ \mathit{true} \rightarrow \mathit{skip}\ \textbf{fi}\ \rangle;\ \langle\ \textbf{if}\ B \wedge C \rightarrow S\ \textbf{fi}\ \rangle \\ \sqsubseteq & \quad \{\text{strengthening the guard}\} \\ & \langle\ \textbf{if}\ B \rightarrow \mathit{skip}\ \textbf{fi}\ \rangle;\ \langle\ \textbf{if}\ B \wedge C \rightarrow S\ \textbf{fi}\ \rangle \\ \sqsubseteq & \quad \{\text{correctly annotating}\} \\ & \langle\ \textbf{if}\ B \rightarrow \mathit{skip}\ \textbf{fi}\ \rangle;\ \{B\}\ \langle\ \textbf{if}\ B \wedge C \rightarrow S\ \textbf{fi}\ \rangle \\ \sqsubseteq & \quad \{\text{logic}\} \\ & \langle\ \textbf{if}\ B \rightarrow \mathit{skip}\ \textbf{fi}\ \rangle;\ \{B\}\ \langle\ \textbf{if}\ C \rightarrow S\ \textbf{fi}\ \rangle \end{array}$$

■

Part (ii) follows from (iii) when we interpret (iii) to mean that a program that contains the refined code has the *same* progress properties as the original program. In order to formalise (iii), we conceptualise two programs, one is the original program that consists of component A and all other components, the other is this program, but with A replaced by A', which is obtained from A by replacing the coarse-grained guarded skip by the pair of fine-grained guarded skips. We next show that

PROOF $pc.A = i \leadsto pc.A = j \quad \Rightarrow \quad pc.A' = i \leadsto pc.A' = j$

First, observe that since the two codes $\{B\}\ \langle\ \textbf{if}\ B \wedge C \rightarrow S\ \textbf{fi}\ \rangle$ and $\langle\ \textbf{if}\ C \rightarrow S\ \textbf{fi}\ \rangle$ are equivalent, it is enough to prove that codes

$$i{:}\langle\ \textbf{if}\ B \wedge C \rightarrow S\ \textbf{fi}\ \rangle\ j{:}$$

and

$$i{:}\langle\ \textbf{if}\ B \rightarrow \mathit{skip}\ \textbf{fi}\ \rangle;\quad k{:}\langle\ \textbf{if}\ B \wedge C \rightarrow S\ \textbf{fi}\ \rangle\ j{:}$$

have the same progress properties. Start by assuming that A can pass its guarded skip

$$\begin{array}{ll} & pc.A = i \\ \leadsto & \quad \{\text{assume}\} \\ & pc.A = i \wedge B \wedge C \\ \leadsto & \quad \{\text{assume}\} \\ & pc.A = j \end{array}$$

then A' is guaranteed to reach $A'.k$ because

$$\begin{array}{ll} & pc.A' = i \\ \leadsto & \quad \{pc.A = i \leadsto pc.A = i \wedge B \wedge C\} \\ & pc.A' = i \wedge B \\ \leadsto & \quad \{IF \text{ progress rule, } B \text{ is GC in } A'\} \\ & pc.A' = k \end{array}$$

Further $pc.A' = i \leadsto pc.A' = k$ does not change the state of the rest of the program, because a guard evaluation action can only change the control state of the component in which the action occurs, which is A'. Hence,

CS: When $pc.A = i$ and $pc.A' = k$ the rest of the program containing A is in the same state as the rest of the program containing A'.

and so

$$\begin{array}{ll} & pc.A' = k \\ \leadsto & \quad \{CS,\ \ pc.A = i \leadsto pc.A = i \wedge B \wedge C\} \\ & pc.A' = k \wedge B \wedge C \\ \leadsto & \quad \{CS,\ \ pc.A = i \wedge B \wedge C \leadsto pc.A = j\} \\ & pc.A' = j \end{array}$$ ■

This concludes the proof that A' is no less progressive than A.

PROOF $pc.A' = i \leadsto pc.A' = j \quad \Rightarrow \quad pc.A = i \leadsto pc.A = j$

The above proof show how A' (with $pc.A' = k$) can get ahead of A (with $pc.A = i$) when $\neg C$ is true, but, of course, the action at $A.k$ must wait for the rest of the program to make C true. Since $pc.A' = k \wedge C \Rightarrow pcA' = k \wedge B \wedge C$ by the annotation of A', and by *CS*, the $A.i$ guard is enabled whenever the $A'.k$ guard is, which concludes the proof that A' is no more progressive than A. ■

5 Conclusion

In the context of sequential programs, Hoare [Hoa69] showed how a sequential program could be verified without reference to its operational semantics. Then, in the context of concurrent programs, Owicki and Gries [OG76] showed how safety properties could be verified by adding *interference freedom* conditions to Hoare's logic, but leaving the underlying logic unchanged. Although this modification was small, the Owicki-Gries theory improved on the previously existing global invariant method of [Ash75] because it avoided a *state explosion* problem [dRea01] by decomposing a global invariant into a program annotation [Lam87]. In this paper we have developed this theory further and incorporated a theory of progress into the formalism.

Several event based models exist, such as [CM88, BS89, LT89, Sha93, Lam94], but, as Lamport suggests, proofs in these models can easily be translated from one model to another, and the difference lies in the ease with which a given program can be formalised in a given model. If a target implementation is based on a concurrent sequential program model, then we see no reason why this implementation should be modelled in an event based one. We therefore see one advantage of our approach over these others in the way that it can support a more direct translation of a program design into code.

The extended theory of Owicki and Gries includes a logic of progress, but it is up to us how to make use of it. Our ultimate aim is to integrate this logic into a method of program design (derivation) in the same kind of style as [FvG99]. Early work in this direction is promising, and a more comprehensive example can be found in [GD05], which presents a derivation of Dekker's program for two process mutual exclusion. In a program verification, we do not have the freedom to change a program when a proof does not work out. We are left with the dilemma of not knowing whether the program or the proof is at fault. In this respect, deriving a program that satisfies a specification is certainly superior. [FvG99] have already shown how commonly occurring design patterns can be identified in both programs and their proofs, and how these patterns can be used to shorten proofs. We believe that patterns such as these will emerge with the extended theory as well. It is a case of realising when they do and noting them accordingly.

We note that although *leads-to* is a widely accepted construct for reasoning about progress, it is not without deficiencies. For instance, while *leads-to* can always be used to express the proposition that P will *eventually* be true, by itself it can not express the proposition that P will be true in the *next* program state. [Sha93] hints at the possibility of using auxiliary variables to express the notion of *next* state. Whether greater expressivity of temporal logic can be achieved in the Owicki-Gries theory by combining auxiliary variables and *leads-to* is a topic of further research.

References

[AO91] K. R. Apt and E. R. Olderog. *Verification of sequential and concurrent programs*. Springer-Verlag New York, Inc., 1991.

[AS89] B. Alpern and F. B. Schneider. Verifying temporal properties without temporal logic. *ACM Transactions on Programming Languages and Systems*, 11(1):147–167, 1989.

[Ash75] E. A. Ashcroft. Proving assertions about parallel programs. *JCSS*, 10:110–135, February 1975.

[BS89] R. J. Back and K. Sere. Stepwise refinement of action systems. In *Proceedings of the International Conference on Mathematics of Program Construction, 375th Anniversary of the Groningen University*, pages 115–138, 1989.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design, A Foundation*. Addison-Wesley, 1988.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[Dij82] E. W. Dijkstra. A personal summary of the gries-owicki theory. In *Selected Writings on Computing: A Personal Perspective*. Springer-Verlag, 1982.

[dRea01] W. P. de Roever et al. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 54 edition, 2001.

[FvG99] W. H. J. Feijen and A. J. M. van Gasteren. *On a Method of Multiprogramming*. Springer-Verlag, 1999.

[GD05] D. Goldson and B. Dongol. Concurrent program design in the extended theory of Owicki and Gries. In M. Atkinson and F. Dehne, editors, *To appear in Proceedings of the Computing: The Australasian Theory Symposium (CATS 2005)*, volume 41, Newcastle, Australia, 2005. Conferences in Research and Practice in Information Technology.

[GP89] R. Gerth and A. Pnueli. Rooting unity. In *Proceedings of the 5th international workshop on software specification and design*, pages 11–19, Pittsburgh, Pensylvania, USA, 1989. ACM Press.

[Gum99] H. P. Gumm. Generating algebraic laws from imperative programs. *Theoretical Computer Science*, 217:385–405, 1999.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Lam87] L. Lamport. Control predicates are better than dummy variables for reasoning about program control. *ACM Transactions on Programming Languages and Systems*, 10(2):267–281, 1987.

[Lam94] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[LT89] N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, 1989.

[LV95] Nancy Lynch and Frits Vaandrager. Forward and backward simulations I: Untimed systems. *Inf. Comput.*, 121(2):214–233, 1995.

[Mis01] J. Misra. *A Discipline of Multiprogramming*. Springer-Verlag, 2001.

[OG76] S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, 1976.

[Sch97] F. B. Schneider. *On Concurrent Programming*. Springer-Verlag, 1997.

[SdR94] F. Stomp and W. P. de Roever. A principle for sequential reasoning about distributed systems. *Formal Aspects of Computing*, 6(6):716–737, 1994.

[Sha93] A. U. Shankar. An introduction to assertional reasoning for concurrent systems. *ACM Computing Surveys*, 25(3):225–262, 1993.

Appendix A: Derived rules of the logic of progress

The logic of UNITY in [CM88, Mis01] is based on an inductive definition of a relation *leads-to*. Given this definition, a number of derived properties are proved. The purpose of this appendix is to confirm that these are also derived rules of our progress logic too. The Inductive Progress Rules in our definition of *leads-to* (in Section 4) are identical to those in [CM88], only the Immediate Progress Rule is different, to take account of the different programming models. Therefore, in what follows the proof of a derived rule will assume that a use of *leads-to* always results from a use of the Basic Progress Rule.

Implication Theorem

$$(P \Rightarrow Q) \Rightarrow (P \leadsto Q)$$

PROOF First note that, for any R, $(P \Rightarrow Q) \Rightarrow (P \wedge \neg Q) \Rightarrow R$. It follows that the three premisses of the Immediate Progress Rule are true on account of this equation, because

1. P **un** Q
   $\equiv$ {By definition, for any atomic statement $\{U\}\ S\}$
   $P \wedge \neg Q \wedge U \Rightarrow wlp.S.(P \vee Q)$
2. $P \wedge \neg Q \Rightarrow pc = i$
3. Any atomic statement can be chosen on account of $P \wedge \neg Q \equiv false$. ∎

Impossibility Theorem

$$(P \leadsto false) \Rightarrow \neg P$$

PROOF First note that, for any S, $wlp.S.false \equiv false$. We look at the three forms of atomic statement that occur in premiss (3) of the Immediate Progress Rule.

1. If $(P \leadsto false)$ because of a *skip* or assignment statement S then

   $P \Rightarrow wlp.S.false$
   $\equiv$ {By wlp and logic}
   $\neg P$

2. If $(P \leadsto false)$ because of an *IF* statement of the form **if** $B_1 \rightarrow S_1 \ [] \ B_2 \rightarrow S_2$ **fi** then

   $(P \wedge B_1 \Rightarrow false) \wedge (P \wedge B_2 \Rightarrow false)$
   $\equiv$ {By logic}
   $(\neg P \vee \neg B_1) \wedge (\neg P \vee \neg B_2)$
   $\equiv$ {By logic}
   $\neg P \vee (\neg B_1 \wedge \neg B_2)$
   $\Rightarrow$ {By premiss (3bi), $P \Rightarrow B_1 \vee B_2$}
   $\neg P$

3. If $(P \leadsto false)$ because of a *DO* statement of the form **do** $B \rightarrow S$ **od** then

   $(P \wedge B \Rightarrow false) \wedge (P \wedge \neg B \Rightarrow false)$
   $\equiv$ {By logic}
   $(\neg P \vee \neg B) \wedge (\neg P \vee B)$
   $\equiv$ {By logic}
   $\neg P \vee (\neg B \wedge B)$
   $\equiv$ {By logic}
   $\neg P$

4. If $(P \leadsto \mathit{false})$ because of i: $\langle S \rangle\ j$: the result follows as $wp.S.\mathit{false} \equiv \mathit{false}$ for any S. ∎

Disjunction Theorem

$$(\forall\, m : m \in W :\ P.m \leadsto Q.m) \Rightarrow ((\exists\, m : m \in W :\ P.m) \leadsto (\exists\, m : m \in W :\ Q.m))$$

Proof As in [CM88] ∎

Cancellation Theorem

$$(P \leadsto Q \vee B) \wedge (B \leadsto R) \Rightarrow (P \leadsto Q \vee R)$$

Proof As in [CM88] ∎

PSP Theorem

$$(P \leadsto Q) \wedge (R \textbf{ un } B) \Rightarrow (P \wedge R \leadsto (Q \wedge R) \vee B)$$

Proof We assume the antecedent and show that, for the consequent, the three premisses of the Immediate Progress Rule are true. The proof uses two equations

(1) $\quad R \wedge \neg Q \wedge \neg B \quad \equiv \quad R \wedge \neg((Q \wedge R) \vee B)$

(2) $\quad Q \wedge (R \vee B) \quad \Rightarrow \quad (Q \wedge R) \vee B$

1. $(P \leadsto Q) \wedge (R \textbf{ un } B)$
   $\Rightarrow$ {By Immediate Progress Rule}
   $(P \textbf{ un } Q) \wedge (R \textbf{ un } B)$
   $\equiv$ {By definition of **un**, for any atomic statement $\{U\}\ S\}$
   $(P \wedge \neg Q \wedge U \Rightarrow wlp.S.(P \vee Q)) \wedge (R \wedge \neg B \wedge U \Rightarrow wlp.S.(R \vee B))$
   $\equiv$ {By logic}
   $P \wedge \neg Q \wedge R \wedge \neg B \wedge U \Rightarrow wlp.S.((P \vee Q) \wedge (R \vee B))$
   $\equiv$ {By logic}
   $P \wedge \neg Q \wedge R \wedge \neg B \wedge U \Rightarrow wlp.S.((P \wedge R) \vee (Q \wedge R) \vee (P \wedge B) \vee (Q \wedge B))$
   $\Rightarrow$ $\{(P \wedge B) \vee (Q \wedge B) \Rightarrow B\}$
   $P \wedge \neg Q \wedge R \wedge \neg B \wedge U \Rightarrow wlp.S.((P \wedge R) \vee (Q \wedge R) \vee B)$
   $\equiv$ {By logic}
   $P \wedge R \wedge \neg((Q \wedge R) \vee B) \wedge U \Rightarrow wlp.S.((P \wedge R) \vee (Q \wedge R) \vee B)$
   $\equiv$ {By definition of **un**}
   $P \wedge R \textbf{ un } (Q \wedge R) \vee B$

2. $P \leadsto Q$
   $\Rightarrow$ {By Immediate Progress Rule}
   $P \wedge \neg Q \Rightarrow pc = i$
   $\Rightarrow$ {By logic}
   $P \wedge R \wedge \neg((Q \wedge R) \vee B) \Rightarrow pc = i$

3. (a) If $(P \leadsto Q)$ because of a *skip* or assignment statement $i\colon S\ j$: then

$(P \rightsquigarrow Q) \wedge (R \textbf{ un } B)$
$\Rightarrow$ {By definition of **un** and $\rightsquigarrow$}
$(P \wedge \neg Q \Rightarrow wlp.(i{:}\, S\ j{:}\,).Q) \wedge (R \wedge \neg B \Rightarrow wlp.(i{:}\, S\ j{:}\,).(R \vee B))$
$\Rightarrow$ {By logic}
$P \wedge \neg Q \wedge R \wedge \neg B \Rightarrow wlp.(i{:}\, S\ j{:}\,).(Q \wedge (R \vee B))$
$\Rightarrow$ {By logic}
$P \wedge R \wedge \neg((Q \wedge R) \vee B) \Rightarrow wlp.(i{:}\, S\ j{:}\,).((Q \wedge R) \vee B)$

(b) If $(P \rightsquigarrow Q)$ because of an *IF* statement of the form $i{:}\,\textbf{if } B_1 \rightarrow j{:}\, S_1 \,[]\, B_2 \rightarrow k{:}\, S_2 \textbf{ fi } l{:}$ then, for premiss (3bi)

$P \rightsquigarrow Q$
$\Rightarrow$ {By definition}
$P \wedge \neg Q \Rightarrow B_1 \vee B_2$
$\Rightarrow$ {By logic}
$P \wedge R \wedge \neg((Q \wedge R) \vee B) \Rightarrow B_1 \vee B_2$

and for premiss (3bii)

$(P \rightsquigarrow Q) \wedge (R \textbf{ un } B)$
$\Rightarrow$ {By definition}
$(R \wedge \neg B \wedge B_1 \Rightarrow (R \vee B)[pc{:=}\, j]) \wedge (R \wedge \neg B \wedge B_2 \Rightarrow (R \vee B)[pc{:=}\, k]) \wedge$
$(P \wedge \neg Q \wedge B_1 \Rightarrow Q[pc{:=}\, j]) \wedge (P \wedge \neg Q \wedge B_2 \Rightarrow Q[pc{:=}\, k])$
$\Rightarrow$ {By logic}
$(P \wedge \neg Q \wedge R \wedge \neg B \wedge B_1 \Rightarrow (Q \wedge (R \vee B))[pc{:=}\, j]) \wedge$
$(P \wedge \neg Q \wedge R \wedge \neg B \wedge B_2 \Rightarrow (Q \wedge (R \vee B))[pc{:=}\, k])$
$\Rightarrow$ {By logic}
$(P \wedge R \wedge \neg((Q \wedge R) \vee B) \wedge B_1 \Rightarrow ((Q \wedge R) \vee B)[pc{:=}\, j]) \wedge$
$(P \wedge R \wedge \neg((Q \wedge R) \vee B) \wedge B_2 \Rightarrow ((Q \wedge R) \vee B)[pc{:=}\, k])$

(c) The case where $(P \rightsquigarrow Q)$ because of a *DO* statement is similar to case (b).

(d) The case where $(P \rightsquigarrow Q)$ because of a $i{:}\,\langle S\rangle\ j{:}$ is similar to case (a). ∎

Induction Theorem Let M be a total function from program states to set W. Let $(W, <)$ be well-founded. Variable m in the following premiss ranges over W and predicates P and Q do not contain free occurrences of variable m.

$$(\forall\, m ::\ P \wedge M = m \rightsquigarrow (P \wedge M < m) \vee Q) \Rightarrow (P \rightsquigarrow Q)$$

Proof As in [CM88] ∎

Completion Theorem Let $P.i$ and $Q.i$ be predicates where i ranges over a finite set.

$$(\forall\, i ::\ (P.i \rightsquigarrow Q.i \vee B) \wedge (Q.i \textbf{ un } B)) \Rightarrow ((\forall\, i ::\ P.i) \rightsquigarrow (\forall\, i ::\ Q.i) \vee B)$$

Proof As in [CM88] ∎